Multiprocessors

To take advantage of a MIMD multiprocessor with n processors, we must usually have at least n threads or processes to execute. The independent threads within a single process are typically identified by the programmer or created by the operating system. At the other extreme, a thread may consist of a few tens of iterations of a loop, generated by a parallel compiler exploiting data parallelism in the loop.

Although the amount of computation assigned to a thread, called the *grain size*, is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction from instruction-level parallelism is that TLP is identified **at a high level by the software or system programmer** and that the threads consist of **hundreds to millions of instructions** that may be executed in parallel.

Threads can also be used to exploit data-level parallelism, although the overhead is likely to be higher than would be seen with a SIMD processor or with a GPU.

Classes of multiprocessors

Existing shared-memory multiprocessors fall into two classes, depending on the memory organisation they apply in their architecture.

Uniform memory access (UMA) multiprocessors

These types of processors have different names they go by:

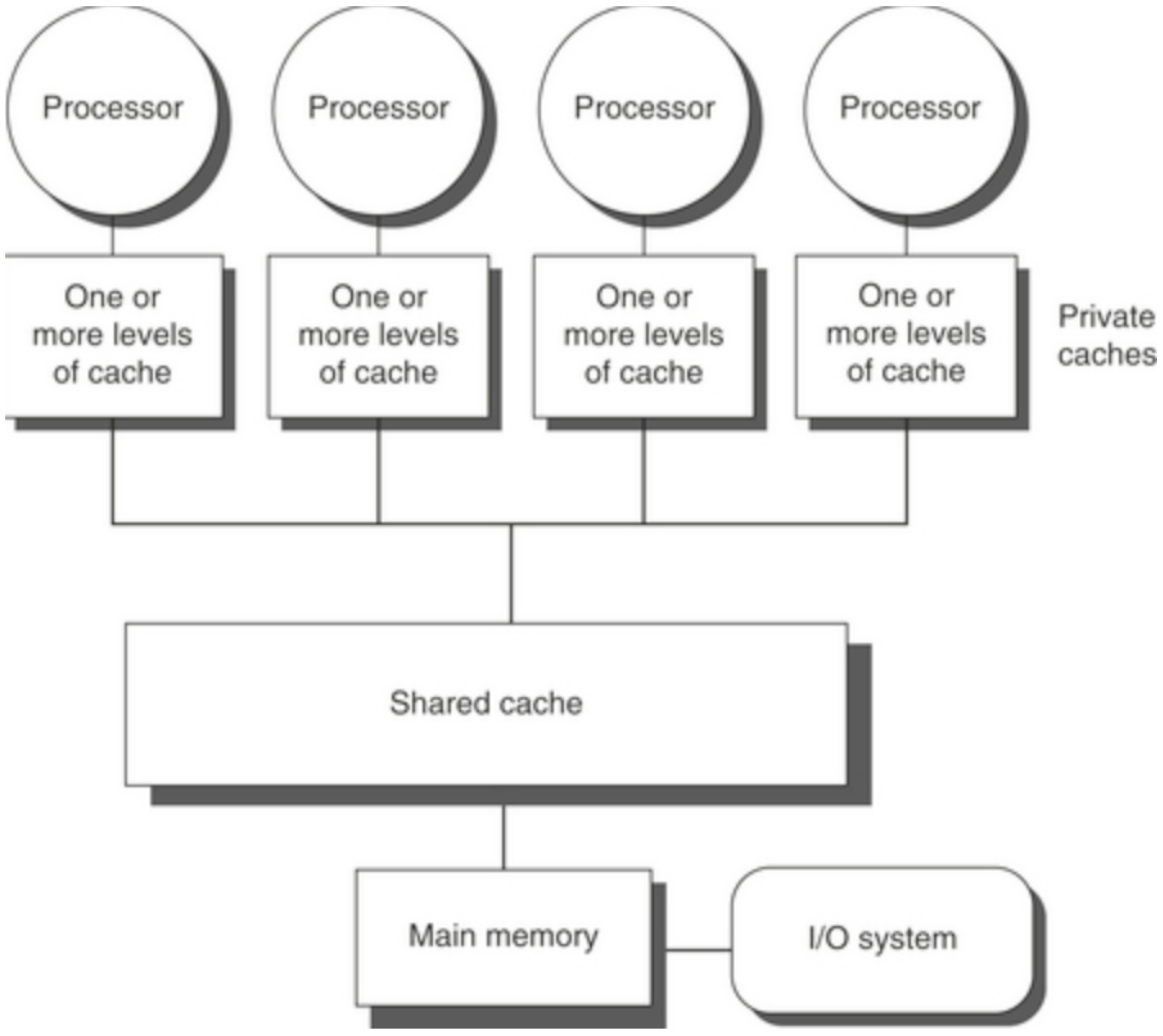
- Symmetric shared-memory multiprocessors (SMPs)
- Centralised shared-memory multiprocessors
- Uniform memory access (UMA) multiprocessors

They typically feature a small number of cores—eight or fewer. For multiprocessors with such small processor counts, it is possible to share a single centralised memory that all processors have equal access to, hence the term *symmetric*.

In multicore chips, the memory is effectively shared in a centralised fashion among the cores, and all existing multicores are SMPs. When more than one multicore is connected, there are separate memories for each multicore, so the memory is distributed, rather than centralised.

These architectures are also called *uniform memory access* because all processors have a uniform latency from memory, even if it is organised in multiple banks.

The following picture represents the high-level topology of processors of this kind.


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

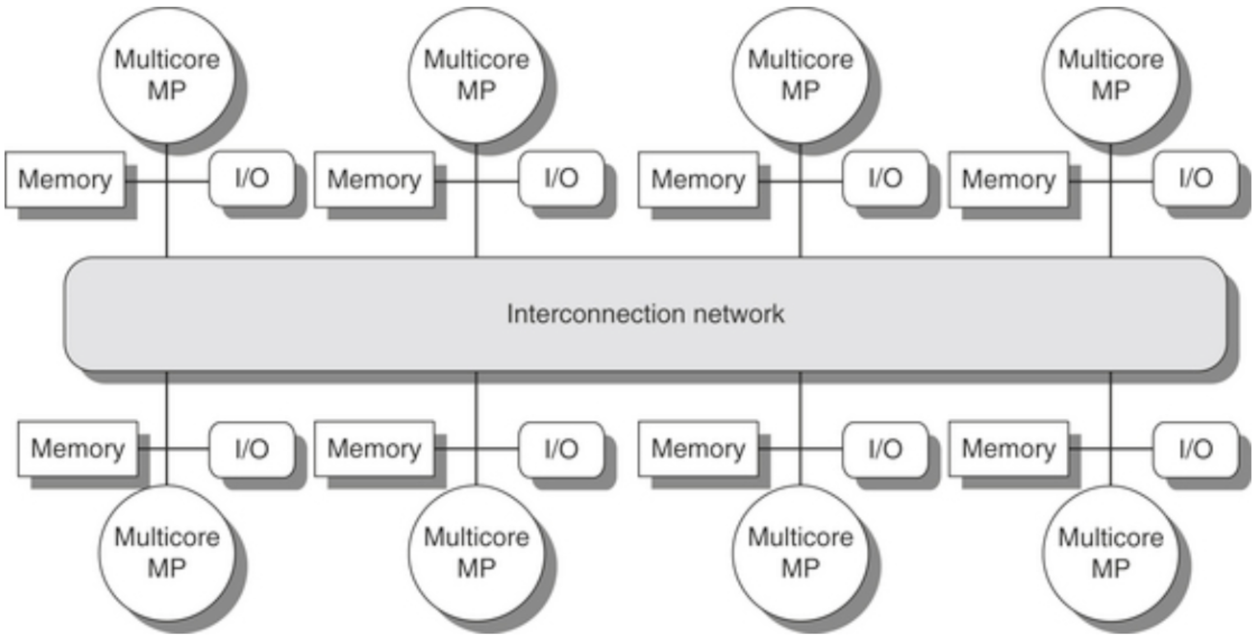
Non-uniform memory access (NUMA) multiprocessors

The alternative design approach consists of multiprocessors with physically distributed memory, called **distributed shared memory** (DSM).

To support larger processor counts, memory must be distributed among the processors rather than centralised; otherwise, the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency.

With the rapid increase of processor performance, the size of a multiprocessor for which distributed memory is preferred continues to shrink. The introduction of multicore processors has meant that even two-chip multiprocessors use distributed memory.

The larger number of processors also raises the need for a high-bandwidth interconnect. Both directed networks and indirect networks are used.



A distributed memory access multiprocessor is also called **non-uniform memory access** or **NUMA**, since the access time depends on the location of a data word in memory.

The key disadvantages for a DSM are that communicating data among processors becomes somewhat more complex, and a DSM requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories.

Shared memory

In both SMP and DSM architectures, communication among threads occurs through a shared address space: a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. The term *shared memory* associated to both architectures refers to the fact that the *address space* is shared, not to the physical topology of memory in the multiprocessor.

Challenges of parallel processing

The application of multiprocessors ranges from running independent tasks with no communication to running parallel tasks where threads must communicate to complete the task. Two important hurdles, both explainable with Amdahl's law, make parallel processing challenging: the first hurdle has to do with the **limited parallelism available in programs**, while the second arises from the **relatively high cost of communications**.

Another major challenge in parallel processing involves the **large latency of remote access** in a parallel processor. In existing shared-memory multiprocessors, communication of data between separate cores may cost 35 to 50 clock cycles and among cores on separate chips anywhere from 100 to 500 clock cycles or more.

These problems are the two biggest performance challenges in using multiprocessors. The problem of inadequate application parallelism must be attacked **primarily in software** with new algorithms that offer better parallel performance. Reducing the impact of long remote latency, instead, can be attacked **both by the architecture and by the programmer**.

In the rest of this chapter, we will focus on techniques for reducing the impact of long remote communication latency.